Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación
IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

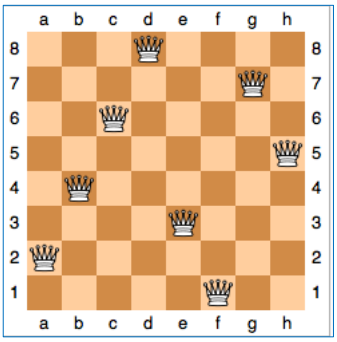
Tema 10 – Recursión Parte 4
+++++ Más Ejemplos

Las 8 Reinas

El Juego de las Ocho-reinas

- Ubicar **8 reinas** en un tablero de **8x8** para que no se ataquen entre ellas.
  - (Es decir, no pueden compartir ni fila, ni columna, ni diagonal)
  - Asumir que existe una función `valido(tablero)` que retorna **True** si hay 1 o menos reinas por fila, columna y diagonal.
- (Hay 4.426.165.368 posibles maneras de ubicar **8 reinas**, pero solo **92** soluciones)
- Y el problema es mucho más complicado (para un "humano común") si hablamos de **n-reinas** (e.g. 20) en un tablero de **n x n**.

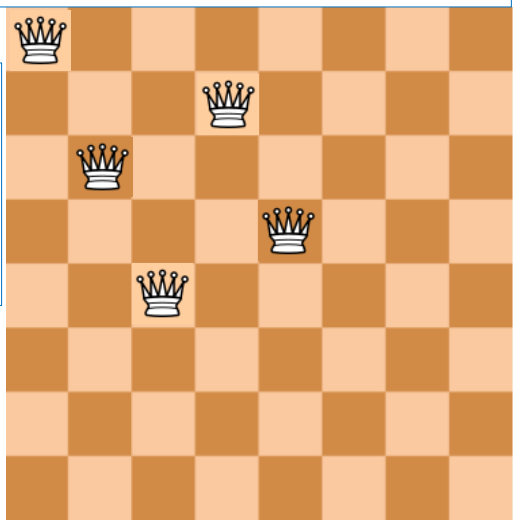
Pero es "*pan comido*" para un alumno de iic1103.



Solución con algoritmo de Back-Tracking.

Ubicando a las reinas…

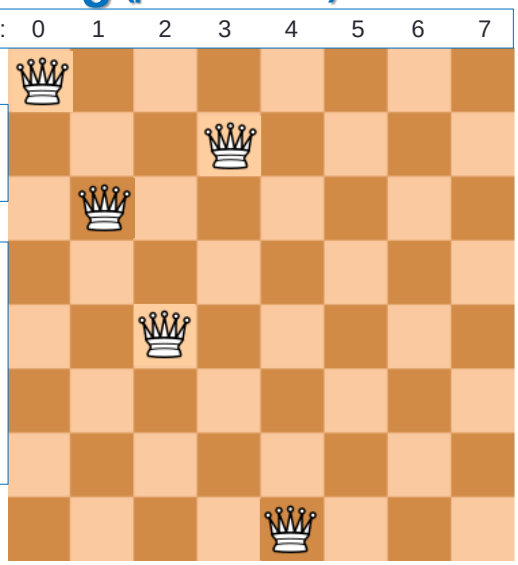
Columnas: 0 1 2 3 4 5 6 7



Idea: Poner una reina en cada columna, partiendo por la columna 0.
Para cada columna, partir en fila 0.
Si no es válido, intentar siguiente fila.

No hay una posición válida en columna 5. Debemos retornar e intentar otro camino.

Backtracking (por columnas)



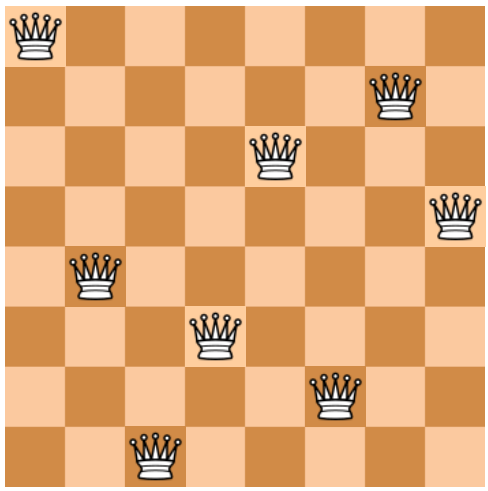
Volvemos a la columna 4 e intentamos en la siguiente fila válida.

Intentamos nueva-mente en columna 5, pero tampoco hay una posición válida. Tenemos que retornar ahora a la columna 3.

Y así sucesivamente…

Solución con Backtracking (por columnas)

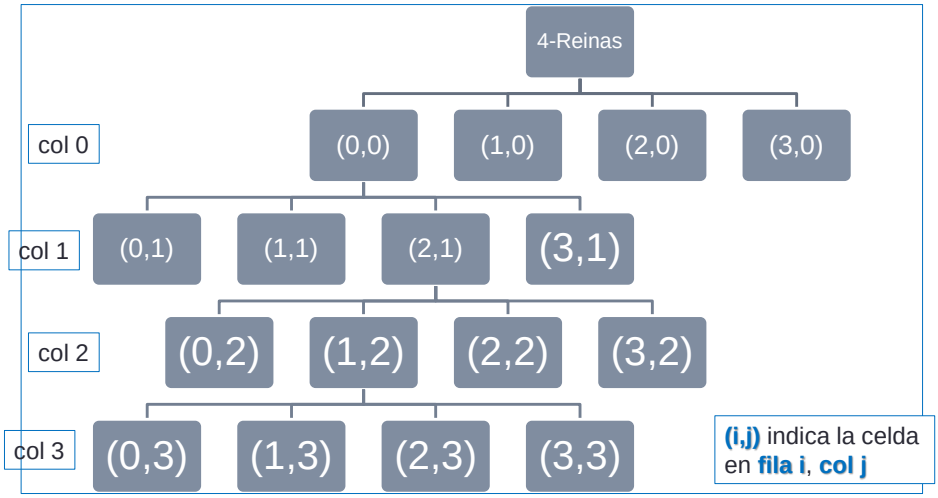
Después de muchas recursiones… tenemos una solución.



n-reinas: Algoritmo de Back-Tracking

- **Caso Base:** Si es válido poner la reina en la última columna → fin! ☺☺☺☺☺
- **Recursión BT:** Ubicar una reina en (alguna fila de) la columna actual (partiendo por columna 0).
  - Si la posición es válida, resolver el problema desde la **columna+1** en adelante.
  - Si la posición es inválida:
    - Probar en la siguiente fila de la columna actual.
    - Si se acabaron las filas, volver a la columna anterior y probar en siguiente fila.

4-Reinas: visualización como árbol (incompleto)



Solución con **BackTracking**: iterar para ir hacia el lado (horizontal), recursión para bajar por las ramas del árbol.

n-Reinas Solución

```
def ubicarReina(tablero,col):
    """ Caso Base """
    if col == len(tablero):
        return True    # listo!
    """ Recursión BT """
    for i in range(len(tablero)):
        tablero[i][col] = True
        if valido(tablero):
            if ubicarReina(tablero,col+1):
                return True
        tablero[i][col] = False
    return False
```

n-Reinas: Variación para imprimir los pasos

```
def ubicarReina(tablero,col,prefijo=""):
    """ Caso Base """
    if col == len(tablero):
        print("\n \n lo logramos!!!")
        return True    # listo!
    """ Recursión BT """
    for i in range(len(tablero)):
        tablero[i][col] = True
        print("\n",prefijo+"probar reina ",col," en ",i,col,end="")
        if valido(tablero):
            if ubicarReina(tablero,col+1,prefijo+"  "):
                return True
        tablero[i][col] = False
        print(prefijo+" no es válida esta posición de reina ",col)
    print(prefijo+"ninguna de las posiciones en columna ",col," es válida")
    return False
```

n-Reinas: función para validar un tablero

Hemos asumido que tenemos una función **`valido(tablero)`** que retorna **True** si hay 1 o menos reinas por fila, columna y diagonal, para un tablero de tamaño n x n. Retorna **False** en caso contrario.

Para entender mejor el problema, construyamos primero una función que imprima todas las diagonales. Pensemos primero en un tablero de 4 x 4 y luego lo extendemos a uno n x n.

```
Los índices de las celdas en el tablero 4 x 4:
( 0 0 ) ( 0 1 ) ( 0 2 ) ( 0 3 )
( 1 0 ) ( 1 1 ) ( 1 2 ) ( 1 3 )
( 2 0 ) ( 2 1 ) ( 2 2 ) ( 2 3 )
( 3 0 ) ( 3 1 ) ( 3 2 ) ( 3 3 )
```

Identifiquemos ahora todas las diagonales (con al menos dos celdas) en el sentido **** y en el sentido **/** :

n-Reinas: función para imprimir diagonales

Las diagonales en el sentido **** son las siguientes:

```
diagonales \
( 0 0 ) ( 1 1 ) ( 2 2 ) ( 3 3 )
( 0 1 ) ( 1 2 ) ( 2 3 )
( 0 2 ) ( 1 3 )
( 1 0 ) ( 2 1 ) ( 3 2 )
( 2 0 ) ( 3 1 )
```

Son 10 diagonales en el caso **4 x 4**. ¿Cuántas son en el caso **8 x 8**? Son 26 y nuestro algoritmo tiene que funcionar para un tablero de **n x n**.

Las diagonales en el sentido **** son las siguientes:

```
diagonales /
( 0 3 ) ( 1 2 ) ( 2 1 ) ( 3 0 )
( 0 2 ) ( 1 1 ) ( 2 0 )
( 0 1 ) ( 1 0 )
( 1 3 ) ( 2 2 ) ( 3 1 )
( 2 3 ) ( 3 2 )
```

n-Reinas: función para imprimir diagonales

Pensemos ahora un algoritmo no-recursivo para imprimir las diagonales **** primero en caso 4 x 4 y luego para un tablero **T** de **n x n** .

Supongamos que las filas las representamos por **f** y las cols por **c** .
La celda en la posición **(f,c)** corresponde al elemento **T [f] [c]** de la lista que representa al tablero. En el caso 4 x 4:

Si hacemos un ciclo con **c** partiendo en 0 y terminando en 2:

- para **c == 0** La diagonal (0 0) (1 1) (2 2) (3 3) la generamos con **T [f] [f+c]** y el **f** variando de 0 a 3
- para **c == 1** La diagonal (0 1) (1 2) (2 3) la generamos con **T [f] [f+c]** y el **f** variando de 0 a 2
- para **c == 2** La diagonal (0 2) (1 3) la generamos con **T [f] [f+c]** y el **f** variando de 0 a 1

Para n x n, se puede observar que **c** debe variar en el rango **range(len(T)-1)**.
Y para cada valor de **c** , la variable **f** debe variar en el rango **range(len(T)-c)** .

n-Reinas: función para imprimir diagonales

El algoritmo anterior para un tablero **T de n x n** se puede representar en el siguiente código Python.
Notar que las filas se representan por **i** y las columnas por **j** .

```
def diagonales(tablero):
    print()
    print("diagonales \\", end=" ")
    for j in range(len(tablero)-1):
        print()
        for i in range(len(tablero)-j):
            print("(",i,i+j,")", end=" ")
```

n-Reinas: función para imprimir diagonales

Para completar las diagonales en el sentido **** nos faltan las siguientes dos diagonales (en tablero 4 x 4) :

(1 0) (2 1) (3 2)
(2 0) (3 1)

Si hacemos un ciclo con **f** partiendo en 1 y terminando en 2:

para **f == 1** La diagonal (1 0) (2 1) (3 2) la generamos con **T [f+c] [c]** y el **c** variando de 0 a 2

para **f == 2** La diagonal (2 0) (3 1) la generamos con **T [f+c] [c]** y el **c** variando de 0 a 1

Para el caso n x n:
Se puede observar que **f** debe variar en el rango **range(1:len(T)-1)**.
Y para cada valor de **f** , la variable **c** debe variar en el rango **range(len(T)-f)** .

n-Reinas: función para imprimir diagonales

El algoritmo anterior para un tablero **T de n x n** se puede representar en el siguiente código Python. Notar que lo hemos agregado al código anterior para completar la impresión de todas las diagonales **** .
(Las filas se representan por **i** y las columnas por **j** .)

```
def diagonales(tablero):
    print()
    print("diagonales \\", end=" ")
    for j in range(len(tablero)-1):
        print()
        for i in range(len(tablero)-j):
            print("(",i,i+j,")", end=" ")
    for i in range(1,len(tablero)-1):
        print()
        for j in range(len(tablero)-i):
            print("(",i+j,j,")", end=" ")
    print()
```

n-Reinas: función para imprimir diagonales

Ahora podemos hacer un análisis similar para descubrir las diagonales en el sentido **/** . La dificultad aquí es que la iteración se debe efectuar en reversa.

Si usamos un ciclo **for** con un **range** que parte de **0**, deberemos usar una variable auxiliar para revertir la cuenta, tal como lo mostramos en el siguiente código.

(También podríamos utilizar un **range(m,n)** donde **m** es negativo y se va incrementando de uno en uno hasta alcanzar **(n-1)**.)

El siguiente código Python imprime las diagonales en el sentido **/** para un tablero **T de n x n**.

(Las filas se representan por **i** y las columnas por **j** .)

n-Reinas: función para imprimir diagonales

Este código Python se debe insertar al final del código anterior, dentro de la función **diagonales()**, para completar la impresión de todas las diagonales.
(Las filas se representan por **i** y las columnas por **j** .)

```
    print()
    print("diagonales /", end=" ")
    for j in range(len(tablero)-1):
        print()
        col= len(tablero)-1-j
        for i in range(len(tablero)-j):
            print("(",i,col-i,")", end=" ")
    for i in range(1,len(tablero)-1):
        print()
        for j in range(len(tablero)-i):
            col= len(tablero)-1-j
            print("(",i+j,col,")", end=" ")
```

n-Reinas: función para imprimir diagonales

Con este código hemos completado el algoritmo para descubrir todas las diagonales en un Tablero de **n x n**.

Notar que lo podríamos hacer un poco más eficiente si ponemos los ciclos para las diagonales **/** dentro de los ciclos para las diagonales **** . Se acorta el código pero se pierde claridad para alguien que lo lee por primera vez.

Ahora lo podemos aplicar (modificándolo un poquito) para validar las diagonales en el juego de las n-Reinas (además de validar las filas y columnas que es mucho más fácil). Esto lo puedes ver en el programa completo publicado en el sitio web del curso.

Tarea: en vez de validar todo el tablero cada vez que se hace una jugada, se podría validar solo la posición de la reina adicional que se pone en el tablero. (Se supone que las anteriores ya están validadas.)

El Espía: un mensaje encriptado

Nuestro espía infiltrado en campo enemigo, nos ha enviado el siguiente mensaje encriptado:

SEND MORE MONEY

¿Es que realmente está en un apuro económico?

No, no, no. Sabemos que contiene un código secreto que nos permitirá conocer las coordenadas donde se encuentra una bomba nuclear que tenemos que desactivar antes que explote.

¿Cómo lo podemos des-encriptar?

La situación es desesperada y no podemos dejar que explote la bomba nuclear. Entonces le pasaremos el problema a nuestros alumnos de iic1103 y les pediremos que este fin de semana se dediquen a resolverlo… con Back-Traking recursivo.

El Espía: un mensaje encriptado

¿Cómo lo podrán des-encriptar nuestros alumnos?

La clave que habíamos acordado previamente con nuestro Espía es que cada letra representa un número distinto entre 0 y 9. Por ejemplo, podríamos asignar los siguientes números a las letras:

S == 0 **E** == 1 **N** == 2 **D** == 3
M == 4 **O** == 5 **R** == 6 **E** == 1
M == 4 **O** == 5 **N** == 2 **E** == 1 **Y** == 8

Notar que cuando asignamos un número a una letra, esto vale para todas sus ocurrencias en las tres palabras.
Para descubrir los números, la llave acordada es que la suma de las dos primeras palabras debe ser igual al valor de la tercera, esto es:

```
 S E N D        0 1 2 3
+M O R E        4 5 6 1
----------------------------
M O N E Y      0 4 6 8 4
```

Claramente 123 + 4561 no es igual a 45218.
Y no nos sirve la asignación del ejemplo.

El Espía: un mensaje encriptado

Construir entonces una función recursiva en Python que recorra por back-tracking todas las posibles asignaciones de números a las letras en el string:

"SENDMORY" (hemos eliminado las repeticiones)

Notar que este string contiene todas las letras del mensaje sin repetir.

Pensar en un árbol invertido de solución donde cada nodo prueba las distintas posibles asignaciones de número a las letras.

Por ejemplo, el primer nivel del árbol (de arriba hacia abajo) itera entre 0 y 9 para la letra "S".
El segundo nivel prueba todos los posibles números que se pueden asignar a "E" sin considerar el número asignado a "S".
El tercer nivel del árbol prueba todos los posibles números que se pueden asignar a "N" sin considerar los números asignados a "S" y a "E".
Y así sucesivamente.

¿Cuál es el **caso base** cuando se llega a un nodo "sin más ramas"?

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación
IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 4
+++++ Más Ejemplos